UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

Probabilistic Topic Modeling with Latent Dirichlet Allocation on Apache Spark

Autor: Carlos Omar Cortés Hinojosa

Director:	**Dr. Jesús Cerquides Bueno.**
Co-director:	**Joan Capdevila Pujol.**
Realitzat a:	**Departament de matemàtica aplicada i anàlisi.**

Barcelona, **30 de juny de 2016**

Abstract

In a world in which we have access to vast amount of data, it is important to develop new tools that allow us to navigate through it. Probabilistic topic models are statistical methods to analyse text corpora and discover themes that best explain its documents. In this work, we introduce probabilistic topic models with special focus on one of the most common models called Latent Dirichlet Allocation (LDA). To learn LDA model from data, we present two variational inference algorithms for batch and online learning. Both algorithms are implemented on a popular Big Data computing framework known as Apache Spark. We introduce this framework and We study the algorithm scalability and topic coherence in two different news data sets from New York Times and BBC News. The results point out to the need to tune up Apache Spark in order to boost its performance and to the goodness of the resulting topics in the BBC News dataset.

keywords: Probabilistic Topic Models, Latent Dirichlet Allocation, Apache Spark.

Resumen

En un mundo en el que tenemos acceso a gran cantidad de información, es importante desarrollar nuevas herramientas que nos permitan navegarla. Los *Probabilistic Topic Models* son métodos estadísticos que analizan corpus de documentos y desvelan los temas que mejor explica sus documentos. En este trabajo presentamos los *Probabilistic Topic Models* con énfasis especial en uno de modelos más comunes llamado *Latent Dirichlet Allocation* (LDA). Para aprender el model LDA a partir del corpus, presentamos dos algoritmos de inferencia variacional para aprendizaje por lotes y online. Ambos algoritmos están implementados en Apache Spark, un popular *framework* Big Data. Introduciremos este framework y estudiaremos la escalabilidad de los algoritmos y la consistencia de los temas en dos conjuntos de notícias provenientes del New York Times y de BBC News. Los resultados señalan la necesidad de ajustar Apache Spark para incrementar el rendimiento y la buena calidad de los temas obtenidos en el conjunto de notícias de BBC News.

Palabras clave: Probabilistic Topic Models, Latent Dirichlet Allocation, Apache Spark.

Resum

En un món on tenim accés a una gran quantitat d'informació és important desenvolupar noves eines que ens permetin navegar-la. Els *Probabilistic Topic Models* són mètodes estadístics que analitzen corpus de documents i rebel·len els temes que millor expliquen aquests documents. En aquest treball presentem els *Probabilistic Topic Models* amb especial èmfasi en un dels algoritmes més comuns anomenat *Latent Dirichlet Allocation* (LDA). Per aprendre el model a partir del corpus, presentem dos algoritmes de infrència variacional per aprenentatge per lots i online. Ambdós algoritmes estan implementats en Apache Spark, un popular *framework* Big Data. Introduïrem aquest framework i estudiarem l'escalabilitat dels algoritmes i la qualitat dels temes en dos conjunts de notícies provinents del New York Times i de BBC News. Els resultats apunten a la necessitat d'ajustar Apache Spark per incrementar el rendiment i la bona qualitat dels temes obtinguts en el conjunt de notícies de BBC News.

Paraules clau: Probabilistic Topic Models, Latent Dirichlet Allocation, Apache Spark.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction and motivation

Historically, when looking for a specific piece of information, like a scientific article or a news piece, the main problem was gaining access to such information. Today we have instant access to as much data as we could wish for thanks to the Internet, and that has become a problem on its own. Our two main tools to navigate through all the online information nowadays are searches and links. From a few keywords a search engine sends us to some documents related to those keywords where we find links to other related documents.

Although we may think this is a powerful way of working with the data and likely the only we can think of, something is missing. It would be really useful if we were able to explore the data or documents based in its theme. Most of the time we are looking for information about a topic or a subject instead of a keyword. So rather than finding documents through keyword searches alone, imagine being able to search and explore documents based on themes that run through them. We might "zoom in" or "zoom out" to find specific or broader themes or even look at how themes changed through time or how they are connected to each other. It is also worth pointing out that knowing the themes of the documents can also improve the accuracy of the search and it is safe to assume that search engines already use Topic Modeling to some extent, we are just introducing a use case for the set of techniques we will explain and use along this work. Figure 1.1 shows one application of the topic-based navigation we talked about.

Topic models provide a way of making possible the situation we talked about. We will give a more detailed and rigorous definition in the following chapter but from a high level point of view we can define Topic Modeling as a way of identifying hidden patterns in a corpus [1]. For example by assigning a set of topics to each document of the collection. We will need to define what we understand for a topic, but for now we can think as the themes the document talks about. The topics we obtain with a Topic Modeling algorithm can be used for example to summarize, visualize and explore the corpus.

Throughout this work we will focus on the most common Topic Modeling algorithm called Latent Dirichlet Allocation (LDA for short). In broad terms, LDA is an unsuper-

[1] Large collection of documents.

vised generative model that tries to capture the intuition that each document exhibits multiples topics.

One of the main challenges of Topic Modeling is dealing with large amounts of information since for a small amount of documents it is easier to just read and study each one of the documents. That forces us to be aware of the amount of computation power it will need and that is why will analyze how LDA works and scales in a big data ecosystem like Apache Spark.

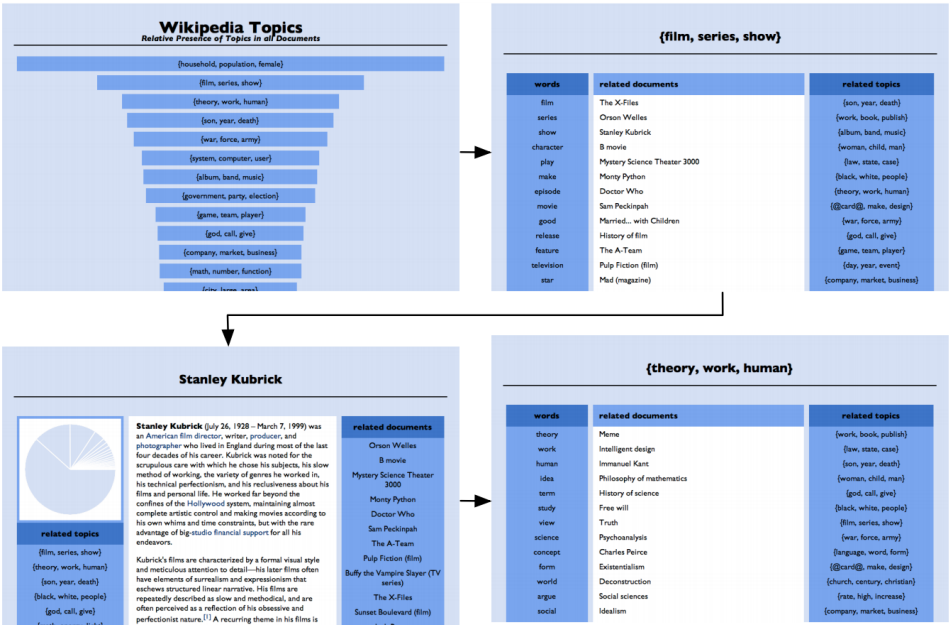


Figure 1.1: Example of the idea we talked about of exploring the data by topic applied to the wikipedia articles. (Source: `http://www.princeton.edu/~achaney/tmve/wiki100k/browse/topic-presence.html`)

1.2 Objectives and organization

This work revolves around two main concepts, Latent Dirichlet Allocation and Apache Spark and the main objectives of this project are to understand and work with those two concepts.

The first main objective of this work is to introduce the concept of Probabilistic Topic Models and present one of the main algorithms in this field called Latent Dirichlet Allocation from a theoretical point of view with two algorithms for the inference process: Batch Variational Bayes and Online Variational Bayes.

As we commented in the previous section, one of the main challenges of Topic Modeling is dealing with large amounts of data so the second main goal of this project is to make use of the Latent Dirichlet Allocation and analyze how it performs and scales in a big data context. Apache Spark plays a big factor in this part so we will need to get a deep understanding of how it works in order to make sense of the performance results we get. For this part we needed a big text dataset, and that is not always easy to find, we

will use a dataset made of news articles from the New York Times[2] that comes already preprocessed as a bag of words and is big enough to allow us analyze the performance.

Lastly, after seeing how it performs and scales we want to show how the results of the Latent Dirichlet Allocation are. By that we mean the topics it finds on the corpus and the topics it assigns to each article. The articles from the New York Times dataset are not available with the dataset, we only have a word count, so we decided to use a second dataset where we have the articles. This dataset is from the BBC[3] and already comes divided in five topics: business, entertainment, politics, sports and tech. This will also help us see how well LDA finds the topics.

According to the objectives this work is organized as follows:

- In Chapter 1 we introduce the work, the objectives and some previous concepts we will need later on.
- In Chapter 2 we review Probabilistic Topic Models with special focus in LDA and the two inference algorithms (Batch Variational Bayes and Online Variational Bayes).
- In Chapter 3 we introduce Apache Spark, explain how it works and all the concepts needed to understand the performance results of Chapter 4.
- In Chapter 4 we test the performance and evaluate the scalability of the LDA implementation in Apache Spark with the New York Times dataset. In the last part of the chapter we show a real word application of LDA using Spark with the BBC news dataset.

Now we will intuitively introduce the two main concepts we previously said: Latent Dirichlet Allocation and Apache Spark.

1.2.1 Latent Dirichlet Allocation

Latent Dirchlet Allocation is a generative model, that means that the algorithm aims to generate a probabilistic model for randomly generating the data. It is easier to understand the idea of a generative model with an example: imagine that we want to learn to distinguish between dogs and cats based on some features of the animal.

One possible approach would be to observe a lot of dogs and cats, that would be our training set, and try to deduce a decision boundary from those observations. This would be the approach used by discriminative algorithms. Instead, a generative algorithm would try to build a model of what dogs look like by looking at dogs and then, by looking at cats, a model of what cats look like. Then to classify a new animal all we need to do is check which one of the two models is more likely to generate the new animal.

The model LDA generates it is not about cats and dogs obviously, it is about documents. The basic idea to generate a document in LDA is as follows:

- We assume we have a number of topics that exists for the whole collection of documents (left side of Figure 1.2).

[2] Available in [1]
[3] Available in [2]

- Randomly select a mixture of topics for the document (the histogram at the right of Figure 1.2). We are assuming we have a number of topics that exists for the whole collection of documents (left side of Figure 1.2).

- For each word in the document we draw a topic from the mixture. Conditioned to the topic, we draw a word (the colored coins of Figure 1.2).

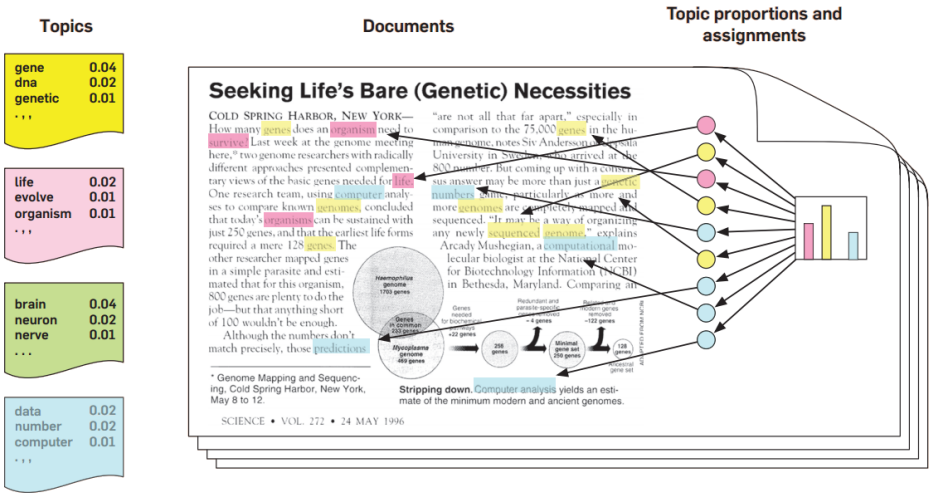


Figure 1.2: The intuitions behind Latent Dirichlet Allocation. All the data shown is only illustrative.
Source: `https://www.cs.princeton.edu/~blei/papers/Blei2012.pdf`.

Of course this generated document will not make any sense to a human, that is not the purpose of the algorithm anyway, we are interested on the way this model can help us gain insights from the document collection. We will see how it does in that regard in Chapter 4.

1.2.2 Apache Spark

Apache Spark is an open source project originally developed at the University of California, Berkeley, in 2009 and currently maintained by the Apache Software Foundation. We previously referred to Spark as a big data environment but more precisely it is a cluster computing framework built around speed, ease of use and sophisticated analytics. It can run in Hadoop clusters through YARN or Spark's standalone mode [4], and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to cover a wide range of workloads including batch processing (similar to Hadoop's MapReduce) and new workloads like streaming, interactive queries, and machine learning. Spark is written and built in Scala, that explains why we have chosen Scala for this project, but it contains APIs for Python, Java and R.

The experimental part of this project has been developed in Spark 1.5.1 that was released in October 2, 2015. Current version as of June 2016 is 1.6.1 where no noticeable changes were done as far as this project concerns.

[4] Standalone mode meaning that the cluster is managed only by Spark but it is still a cluster.

1.3 Planning

This work is worth 18 ECTS credits and each credit corresponds to 25 work hours. So, this work is equivalent to 450 work hours. The usual span of a project like this is one semester (around 20 weeks), but since I was working full-time during the realization of this project it took two semester (around 40 weeks) to finish it.

1.4 Notation and terminology

During this work we will use the same terminology used by David Blei in the 2003 article where he presented the Latent Dirichlet Allocation [5] so we will use the language of text collections. Formally, we define the following terms:

- A word is the basic unit of discrete data, defined to be an item from a vocabulary indexed by $\{1, ..., V\}$. We represent words using unit-basis vectors that have a single component equal to one and all other components equal to zero. Thus, using superscripts to denote components, the v-th word in the vocabulary is represented by a by a V-vector w such that $w^v = 1$ and $w^u = 0$ for $u \neq v$.
- A document is a sequence of N words denoted by $\mathbf{w} = (w_1, w_2, ..., w_N)$, where w_n is the n-th word in the sequence.
- A corpus is a collection of M documents denoted by $D = w_1, w_2, ..., w_M$.

1.4.1 Some Probability Concepts

Prior and posterior distributions

When introducing the Latent Dirichlet Allocation in chapter 2, we will be using the terms *prior* and *posterior* probabilities and *likelihood* function:

- The prior probability distribution of and uncertain quantity is the probability distribution that express the value of this quantity before some evidence is taken into account.
- The posterior probability is the conditional probability after some evidence is taken into account.
- The likelihood function of a set of parameters given some outcome is equal to the probability of observing said outcome given those parameter values.

If the prior and the posterior distributions are in the same distribution family, then the prior and posterior are called *conjugate* distributions and the prior is called a conjugate prior for the likelihood funtion.

[5] Article can be found in [6].

Given a prior $p(\theta)$ and observations x with the likelihood $p(x \mid \theta)$, the posterior is defined as:

$$p(\theta \mid x) = \frac{p(x \mid \theta)p(\theta)}{p(x)}$$

Jensen Inequality

In the context of probabilities, the Jensen Inequality is stated as follows: If X is a random variable and f a convex function then:

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$$

And, if f is a concave function the inequality turns:

$$f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)]$$

Kullback-Leibler (KL) divergence

When formulating the optimization problem in Latent Dirichlet Allocation we will use the KL divergence to express the optimization problem. The KL divergence is a measure of the difference between two probability distributions P and Q. It is important to note that the KL divergence is not symmetrical in P and Q so it's not a metric.

The KL divergence can be interpreted as a way to express the amount of information lost when Q is used to approximate P [6]

Given two continuous distributions P and Q the KL divergence is defined as follows:

$$D_{KL}(P \mid\mid Q) = \int_{-\infty}^{+\infty} p(x) log \frac{p(x)}{q(x)} dx$$

This can also be expressed using the definition of expectation (and expressing the log of a division as difference of log) as:

$$D_{KL}(P \mid\mid Q) = \mathbb{E}_p[log\ p(x)] - \mathbb{E}_p[log\ q(x)]$$

This is the expression we will use later on instead of the definition.

We present now the two probability distributions we will be using when we define the different models in this work: the Multinomial and the Dirichlet distributions. **Multinomial Distribution**

The multinomial distribution models the histogram of outcomes of an experiment with k possible outcomes that is performed n times. The parameters of this distribution are the

[6] This is a quote from the book Model Selection and Multi-Model Inference by K.P. Burnham and D.R. Anderson

probabilities of each of the k possible outcomes occurring in a single trial, $p_1, ..., p_k$. This is a discrete, multi-variate distribution with probability mass function:

$$f(x_1, \ldots, x_k \mid n, p_1, \ldots, p_k) = \Pr(X_1 = x_1 \text{ and } \ldots \text{ and } X_k = x_k) =$$

$$= \begin{cases} \dfrac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}, & \text{when } \sum_{i=1}^{k} x_i = n \\ \\ 0 & \text{otherwise} \end{cases}$$

A good example of how the multinomial distribution is used is to think of rolling a dice several times. Say you want to know the probability of rolling 3 fours, 2 sixes, and 4 ones in 9 total rolls given that you know the probability of landing on each side of the dice. The multinomial distribution models this probability.

For LDA we will use a Multinomial Distribution with $n = 1$ that is also known as Categorical Distribution that simplifies the probability mass function to:

$$f(x_1, \ldots, x_k \mid p_1, \ldots, p_k) = \prod_{i=1}^{k} p_i^{x_i}$$

Dirichlet Distribution

As we can guess by its name, we will be using this distribution when we explain the Latent Dirichlet Allocation. The Dirichlet distribution is a bit tricky to understand because it deals with distributions over the probability simplex. This means that the Dirichlet distribution is a distribution over discrete probability distributions. Its probability mass function is (maintaining the notation introduced for the Multinomial definition):

$$P(P = \{p_i\} | \alpha_i) = \frac{\prod_i \Gamma(\alpha_i)}{\Gamma(\sum_i \alpha_i)} \prod_i p_i^{\alpha_i - 1}$$

Where $\alpha = (\alpha_1, \cdots, \alpha_k)$ with $\alpha_i > 0$ for each i is the parameter of the Dirichlet distribution.

The key to understanding the Dirichlet distribution is realizing that instead of it being a distribution over events or counts (as most common probability distributions are), it is a distribution over discrete probability distributions (eg. multinomials).

The reason why we use the Dirichlet Distribution in Latent Dirichlet Allocation is that we will model the topics as multinomials and the Dirichlet Distribution is the conjugate prior of the multinomial which facilitates the development of the algorithm.

We call exchangeable Dirichlet Distribution with a single scalar parameter η to a Dirichlet distribution where $\alpha_i = \eta$ for all i.

1.4.2 Graphical Model

Graphical models are a way to encode the conditional dependence structure between random variables of probabilistic models in a visual way (in a directed graph):

- Nodes represent random variables.
- Edges denote possible dependence.
- Shaded variables are shaded nodes.
- Plates represent replicated structure.

For example the following figure shows an example of a graphical model. The edge between X_1 and Y means that X_1 depends on Y. The shaded nodes X_1, ..., X_n means that they are observed variables. Y is a hidden or latent variable. The plate denote the replicated structure.

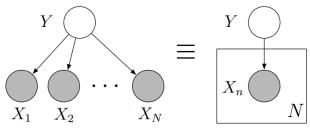


Figure 1.3: Example of a graphical model and the usage of plates.

The structure of the graph defines the pattern of conditional dependence between the random variables. From the graph in the figure we can derive the joint probability distribution:

$$p(Y, X_1, \cdots, X_n) = p(Y) \prod_{n=1}^{N} p(X_n \mid Y)$$

Chapter 2

Probabilistic Topic Models

2.1 Background

We already talked a bit about this concept in the introduction but we have not defined the concept of Probabilistic Topic Model yet. From Blei 2012 "Probabilistic Topic modeling algorithms are statistical methods that analyze the words of the original texts to discover the themes that run through them." In section 1.1.1 we introduced an example of a Probabilistic Topic Model. We will present it again in this chapter along with all the details we ignored in the introduction. But before that we will review some significant progress made in this field before LDA.

In the following sections we use the notation introduced in section 1.4 where $\mathbf{w}$ denotes a document, w_n denotes the n-th word of the document and N denotes the length of the document.

2.1.1 Unigram Model [1]

The most simple Probabilistic Topic Model we are going to talk about is the Unigram Model. In this model the words of every document are generated independently from a single multinomial distribution. So the probability of a document is the product of the probability of each of the words:

$$p(\mathbf{w}) = \prod_{n=1}^{N} p(w_n)$$

We can illustrate the model by the graphical model in figure 2.1.

[1] More detail about the Unigram Model as well as the generalization to the n-gram model can be found in [3]

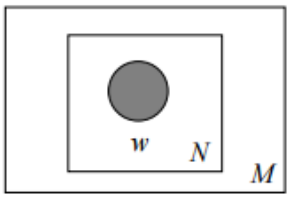


Figure 2.1: Graphical model for the Unigram Model.

2.1.2 Mixture of Unigrams [2]

This model adds a discrete random variable z to the Unigram Model that represents the topic of the documents. To generate a document under this model we select a topic z for the document and then select the words from the conditional Multinomial $p(w|z)$. So the probability of a document is the sum of the probability of choosing the topic z represented as $p(z)$ and then drawing the N words independently conditioned by the chosen topic, that is $p(w \mid z)$:

$$p(\mathbf{w}) = \sum_z p(z) \prod_{n=1}^{N} p(w_n|z)$$

The mixture of Unigrams Model is represented by the following graphical model (Figure 2.2):

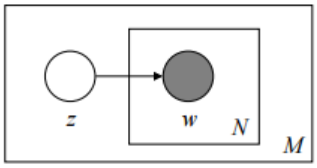


Figure 2.2: Graphical model for the Mixture of Unigrams Model.

When we generate this model from observations of a corpus we can consider the words distributions p(w | z) as representations of topics assuming that each documents is about a single topic. We will face this issue in LDA by introducing one more parameter to the model.

2.1.3 Probabilistic latent Semantic Indexing (PLSI) [3]

PLSI attempts to solve the issue of having only one topic per document. PLSI proposes that a document label d and a word w_n are conditionally independent given and unobserved topic z. And by adding this we are allowing a document to have multiple topics, the topic weights of a documents are represented by $p(z \mid d)$.

$$p(d, w_n) = p(d) \sum_z p(w_n|z)p(z|d)$$

[2] In depth article about the mixture of unigram model can be found in [4]
[3] Original article can be found in [5]

Here, d only acts as index to the document in the collection. d is a Multinomial random variable with as many possible values as the size of the training documents and the model learns the topic weights, $p(z \mid d)$, only for those documents on which it is trained since d is the index of the document in the collection. That makes it hard to assign to new unseen documents and makes the model prone to overfitting the training set.

The PLSI Model can be represented by the following graphical model (Figure 2.3):

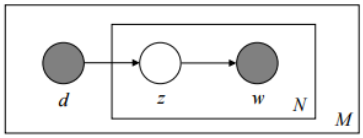


Figure 2.3: Graphical model for the Probabilistic latent Semantic Indexing Model.

2.2 Latent Dirichlet Allocation

In this section we will be following the original article where the Latent Dirichlet Allocation [4]. Our main goals here are to understand the LDA model and understand the setting of the algorithm and how we transform the inference problem to an optimization problem. We consider the notation introduced section 1.3.

We have already introduced the Latent Dirichlet Allocation as a generative probabilistic model of a corpus of text documents and presented in an intuitive manner the generative process. We will give now a more rigorous and in-depth explanation. Documents are represented as mixtures over latent topics and each topic is characterized by a distribution over words. LDA assumes the following generative process for each document:

1. Choose $N \sim Poison(\xi)$. This will represent the length of the document.
2. Choose $\theta \sim Dir(\alpha)$. This will represent the mixture of topics for the document.
3. For each of the N words w_n:
   (a) Choose a topic $z_n \sim Multinomial(\theta)$. The topic we choose w_n from.
   (b) Choose a word w_n from $p(w_n \mid z_n, \beta)$, a multinomial probability conditioned on the topic z_n.

First we observe that the model assumes that the document length, N, is independent of the data generating variables (θ and z) so the Poisson can be changed for another distribution that generates more realistic lengths of documents. In this model we are assuming that the dimensionality k of the Dirichlet distribution is fixed. In practice this will mean that we fix the number of topics before executing the algorithm. The word probabilities are parameterized by a $k \times V$ matrix β where $\beta_{i,j} = p(w^j = 1, z^i = 1)$ is the probability of the word j of the vocabulary in the topic i . For now, we can think of β as a fixed quantity that needs to be estimated. θ in this model represents the probability of each topic for the document.

[4]The Article can be found in [6]

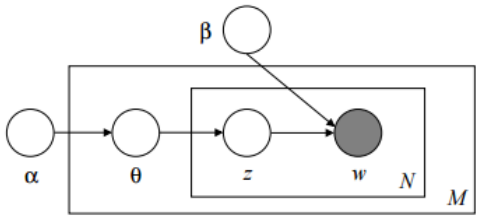


Figure 2.4: Graphical model for the Latent Dirichlet Allocation.

Figure 2.4 shows the probabilistic graphical model for LDA. We can see that there is three level of parameters in LDA, α and β that are corpus-level parameters. θ is a document-level parameter sampled once per document. And finally, z and w are word-level variables and are sampled once for each word in each document.

Given the parameters α and β, the joint distribution of a topic mixture θ, a set of N topics z, and a set of N words $\mathbf{w}$ (a document) is given by:

$$p(\theta, z, \mathbf{w} \mid \alpha, \beta) = p(\theta \mid \alpha) \prod_{n=1}^{N} p(z_n \mid \theta) p(w_n \mid z_n, \beta) \tag{2.1}$$

Observe that $p(z_n \mid \theta)$, the probability of choosing the topic n given a topic mixture θ, is θ_i for the unique i that corresponds to the topic z_n. To obtain the marginal distribution of a document we integrate over θ and sum over z:

$$p(\mathbf{w} \mid \alpha, \beta) = \int p(\theta \mid \alpha) \left(\prod_{n=1}^{N} \sum_{z_n} p(z_n \mid \theta) p(w_n \mid z_n, \beta) \right) d\theta \tag{2.2}$$

If we want to obtain the probability of a corpus we just need to take the product of the marginal probabilities of single documents:

$$p(D \mid \alpha, \beta) = \prod_{d=1}^{D} \int p(\theta_d \mid \alpha) \left(\prod_{n=1}^{N_d} \sum_{z_{dn}} p(z_{dn} \mid \theta_d) p(w_{dn} \mid z_{dn}, \beta) \right) d\theta_d$$

As we earlier commented, the goal of the Latent Dirichlet Allocation is not to generate documents or corpus. The corpus is already generated, what we want is to gain information about the topics of the documents. That brings us our main inference problem, we are interested on computing the posterior probability of the variables θ and z given the data $\mathbf{w}$ and the hyperparameters α and β.

$$p(\theta, z \mid \mathbf{w}, \alpha, \beta) = \frac{p(\theta, z, \mathbf{w} \mid \alpha, \beta)}{p(\mathbf{w} \mid \alpha, \beta)} \tag{2.3}$$

See (2.1) for the value of the numerator and (2.2) for the denominator of the posterior distribution (2.3). The computation of this denominator is intractable to compute in general because when we write equation 2.2 in terms of the model parameters we find an intractable coupling between θ and β [5]. Since this distribution is intractable for exact inference we need to use an approximate inference algorithm. We will show two algorithms

[5]Why this coupling is intractable goes beyond the scope of this project.

for this purpose: Batch Variational Inference, the one introduced in the original article we've been following so far, and Online Variational Inference, introduced in 2010.

2.3 Variational inference for LDA

The intractability of the above equation 2.2 is due to the coupling between variables θ and β. The variational inference scheme considers a factorized variational distribution which approximates the posterior distribution. This factorized variational distribution is expressed as follows:

$$q(\theta, z \mid \gamma, \phi) = q(\theta \mid \gamma) \prod_{n=1}^{N} q(z_n \mid \phi_n) \tag{2.4}$$

Where γ and $(\phi_1, \cdots, \phi_N)$ are the free variational parameters. The resulting graphical model is shown in figure 2.5.

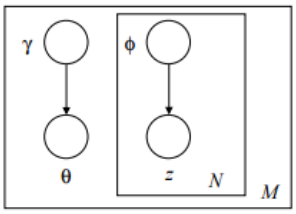


Figure 2.5: Graphical model of the variational distribution.

To determine the values of the variational parameters γ and $(\phi_1, \cdots, \phi_N)$ we set up an optimization problem. We want to maximize the log likelihood of a document which translates into an optimization problem. Now we show the process to obtain the optimization problem.

$$log p(\mathbf{w} \mid \alpha, \beta) \overset{(1)}{=} log \int \sum_z p(\theta, z, \mathbf{w} \mid \alpha, \beta) d\theta \overset{(2)}{=} log \int \sum_z p(\theta, z, \mathbf{w} \mid \alpha, \beta) \frac{q(\theta, z \mid \gamma, \phi)}{q(\theta, z \mid \gamma, \phi)} d\theta \overset{(3)}{=}$$

$$\overset{(3)}{=} log \mathbb{E}_{q(\theta, z \mid \gamma, \phi)} \left[\frac{p(\theta, z, \mathbf{w} \mid \alpha, \beta)}{q(\theta, z \mid \gamma, \phi)} \right] \overset{(4)}{\geq} \mathbb{E}_{q(\theta, z \mid \gamma, \phi)} \left[log \left(\frac{p(\theta, z, \mathbf{w} \mid \alpha, \beta)}{q(\theta, z \mid \gamma, \phi)} \right) \right] \overset{(5)}{=}$$

$$\overset{(5)}{=} \mathbb{E}_{q(\theta, z \mid \gamma, \phi)} \Big[log p(\theta, z, w \mid \alpha, \beta) - log q(\theta, z \mid \gamma, \phi) \Big] \overset{(6)}{=}$$

$$\overset{(6)}{=} \mathbb{E}_{q(\theta, z \mid \gamma, \phi)} \Big[log p(\theta, z, \mathbf{w} \mid \alpha, \beta) \Big] - \mathbb{E}_{q(\theta, z \mid \gamma, \phi)} \Big[log q(\theta, z \mid \gamma, \phi) \Big]$$

(1) we marginalize over θ and z.

(2) we multiply and divide by q.

(3) we use the definition of expectation.

(4) we apply Jensen's inequality ($f(\mathbb{E}[X]) \geq \mathbb{E}[f(x)]$ if $f(\mathbb{E}[X])$ is a convex function).

(5) Apply the property of log function that says that log of the division is equal to the difference of log's.

(6) Since the expectation is a linear operator we can separate the two log.

We obtained a lower bound for the $logp(\mathbf{w} \mid \alpha, \beta)$ that we will denote for now on as $L(\gamma, \phi \mid \alpha, \beta)$. Now we are going to observe that the difference between the lower bound we obtained and $logp(\mathbf{w} \mid \alpha, \beta)$ is the KL divergence between $p(\theta, z \mid \mathbf{w}, \alpha, \beta)$ and $q(\theta, z \mid \gamma, \phi)$:

First we calculate the KL divergence between $p(\theta, z \mid \mathbf{w}, \alpha, \beta)$ and $q(\theta, z)$ using the expression we deduced in the introduction:

$$D_{KL}(q(\theta, z \mid \gamma, \phi) \mid\mid p(\theta, z \mid \mathbf{w}, \alpha, \beta)) = \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logq(\theta, z \mid \gamma, \phi)\Big] - \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logp(\theta, z \mid \mathbf{w}, \alpha, \beta)\Big]$$

Adding the KL divergence with the lower bound we obtain:

$$D_{KL}(q(\theta, z \mid \gamma, \phi) \mid\mid p(\theta, z \mid \mathbf{w}, \alpha, \beta)) + L(\gamma, \phi \mid \alpha, \beta) =$$
$$\mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logq(\theta, z \mid \gamma, \phi)\Big] - \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logp(\theta, z \mid \mathbf{w}, \alpha, \beta)\Big] + \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logp(\theta, z, \mathbf{w} \mid \alpha, \beta)\Big]$$

$$-\mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logq(\theta, z \mid \gamma, \phi)\Big] \overset{(7)}{=}$$

$$\overset{(7)}{=} \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[log\frac{p(\theta, z, \mathbf{w} \mid \alpha, \beta)}{p(\theta, z \mid \mathbf{w}, \alpha, \beta)}\Big] \overset{(8)}{=} \mathbb{E}_{q(\theta,z|\gamma,\phi)}\Big[logp(\mathbf{w} \mid \alpha, \beta)\Big] \overset{(9)}{=} logp(\mathbf{w} \mid \alpha, \beta)$$

(7) We use that the expectation is linear and that difference of log's is equal to the log of the division.

(8) Using equation (2.3).

(9) $logp(\mathbf{w} \mid \alpha, \beta)$ does not depend on q.

So, as a function of the variational distribution, minimizing the KL divergence is the same as maximizing the ELBO. We obtained our optimization problem:

$$(\gamma^*, \phi^*) = argmin_{(\gamma,\phi)} D_{KL}(q(\theta, z \mid \gamma, \phi) \mid\mid p(\theta, z \mid \mathbf{w}, \alpha, \beta))$$

This minimization can be achieved by a fixed point method[6]. This yields algorithm 1:

Where Ψ is the first derivative of the $log\Gamma$ function.

[6]We skip the calculations since they are particular to this application of Variational inference, all the details can be found in the apendix of the original article.

Algorithm 1 Variational Inference for LDA

Initialize $\phi_{ni}^0 := 1/k$ for all i and n
Initialize $\gamma_i := \alpha_i + N/k$ for all i
repeat
    for $n = 1$ to N
        for $i = 1$ to k
            $\phi_{ni}^{t+1} := \beta_{iw_n} exp(\Psi(\gamma_i^t))$
        Normalize ϕ_{ni}^{t+1} to sum 1
    $\gamma^{t+1} := \alpha + \sum_{n=1}^{N} \phi_{ni}^{t+1}$
until convergence

Parameter Estimation

So far we have treated the model parameters α and β as fixed quantities, when using the algorithm for real world applications we want a way to estimate them. Given a corpus of $D = \{\mathbf{w}_1, ..., \mathbf{w}_M\}$ documents, we want the values for α and β that maximize the log likelihood of the corpus:

$$\ell(\alpha, \beta) = \sum_{d=1}^{M} logp(\mathbf{w}_d \mid \alpha, \beta)$$

We already commented on the fact that $p(\mathbf{w} \mid \alpha, \beta)$ can not be computed, but we have a lower bound on the log likelihood that can be maximized with respect to α and β. So we can obtain estimates for the parameters via a *Variational Expectation Maximization* procedure that can be summarized as follows:

- **E-step**: For each document, find the optimizing parameters $\{\gamma_d^*, \phi_d^* : d \in D\}$ as described before.
- **M-step**: Maximize the resulting lower bound with respect to the model parameters α and β.

2.3.1 Smoothed LDA [7]

Before we start explaining the the Online Variational Inference we need to present a slight modification to the LDA model we have shown in the previous section called the smoothed LDA. The smoothed LDA modifies the LDA model by treating β as a random matrix. Each row of β is drawn independently from an exchangeable Dirichlet with parameter η. The graphical model representation for this model is shown in figure 2.6. We add the η hyperparameter because with the LDA model we estimate β assigning probability zero to all the words not found in any document of the training set used to build the model.

The inference process we showed in the previous section can be adapted to bear in mind the fact that β is a random $k \times V$ matrix (or equivalently k vectors of length V), we add a parameter λ to the factorized variational distribution. We will not show the details but a similar process to the one we have shown in the previous section the variational

[7] This model is showed in the original article of LDA found in [6]

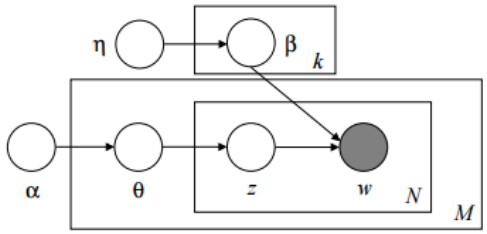


Figure 2.6: Graphical model for the Smoothed Latent Dirichlet Allocation.

process can be adapted to the smoothed model and algorithm 1 remains the same with the addition of the update for the new parameter λ we introduced. The update equation for λ is:

$$\lambda_{ij} = \eta + \sum_{d=1}^{M} \sum_{n=1}^{M_d} \phi^*_{d_{ni}} w^j_{d_n}$$

2.4 Online Variational Inference[8]

Variational inference requires to pass through the whole dataset for each iteration, and that can be a problem when working with big datasets and is not an option to use cases where the data is arriving constantly. The Online Variational Inference algorithm aims to solve both problems without compromising the quality of the obtained topics and with an algorithm nearly as simple as Variational Inference. For this algorithm we will only highlight the differences respect the Variational Inference and the resulting algorithm. The key difference is the way to fit the variational parameter λ respect the Variational Inference explained in the previous section.

As we observe the t-th document (or equivalently a vector of words counts) we perform the E-step like Variational inference[9] to find locally optimal values for γ_t and ϕ_t while holding λ fixed. And then (the M-step) compute the optimal value for λ if the whole document corpus consisted of this t-th document repeated M times (remember that M is the number of documents in the corpus). After we have the optimal value of λ for the document, we call it λ^*, we update the value of λ using a weighted average of its previous value and λ^*_d. The weight given to λ^* is given by $\rho_t = (\tau_0 + t)^{-\kappa}$, with $\kappa \in (0.5, 1]$. κ controls the importance we give to most recents values of λ^*.

Algorithm 2 shows the inference process given a vector of word counts. The explicit formulas for $\mathbb{E}_q[log\theta_{tk}]$ and $\mathbb{E}_q[log\beta_{kw}]$ are:

$$\mathbb{E}_q[log\theta_{tk}] = \Psi(\gamma_{tk}) - \Psi(\sum_{i=1}^{K} \gamma_{ti})$$

$$\mathbb{E}_q[log\beta_{kw}] = \Psi(\lambda_{kw}) - \Psi(\sum_{i=1}^{W} \lambda_{ki})$$

[8]Original article can be found in [8].
[9]The E-step is the same as the E-step of Variational Inference applied to the smoothed LDA model.

Where Ψ denotes de derivative of the logarithm of the gamma function.

Algorithm 2 Online Variational Inference for smoothed LDA

Define $\rho_t \triangleq (\tau_0 + t)^{-\kappa}$
Initialize λ randomly.
for $t = 0$ **to** ∞ **do**
    E-Step:
    Initialize $\gamma_{tk} = 1$ (The constant 1 is arbritary.)
    repeat
        Set $\phi_{twk} \propto exp\{\mathbb{E}_q[log\theta_{tk}] + \mathbb{E}_q[log\beta_{kw}]\}$
        Set $\gamma_{tk} = \alpha + \sum_w \phi_{twk} n_{tw}$
    until $\frac{1}{K}\sum_k \mid$ change in $\gamma_{tk} \mid < 0.00001$
    M-Step:
    Compute $\lambda^*_{kw} = \eta + Dn_{tw}\phi_{twk}$
    Set $\lambda = (1 - \rho_t)\lambda + \rho_t\lambda^*$
end for

The fact that we are updating all the parameters by only observing a single document allows us to use the algorithm in situations where the data is constantly arriving

Chapter 3

Apache Spark

We have already introduced Apache Spark in Chapter 1, in this chapter we will analyze in depth the Apache Spark ecosystem and its architecture but first we comment on the motivation behind Spark and the reasons behind the rise in popularity it is currently experimenting.

3.1 Motivation

Apache Sparks appears around 2009 as a response to the need for tools that adapted efficiently to new workloads (e.g. interactive queries and machine learning) since the options available at the time were not optimized for that[1]. The main tool available at the time was Hadoop's MapReduce that it is a good solution for one-pass computations but not optimized for iterative workloads that are common in machine learning. In the MapReduce framework each step done while processing the data has one Map phase and one Reduce phase and you will need to convert any processing needs into MapReduce pattern to take advantage of this solution adding complexity to the process. In figure 3.1 we can see an example of MapReduce.

The concept behind MapReduce is divide the data in multiple parts and storing them in different machines in a cluster performing the operations on those machines (Mapping phase) and sending the results to another machine in the cluster (Reduce phase). The main performance drawback for iterative workloads comes from the fact that the data is stored and read from the disk in each step of processing and I/O from disk is really slow. Map step reads data from disk/hard drive, processes it and send it back to disk before performing shuffling operation to send data to reduce. At reduce, again reads data from disk, processes it and writes data back to disk. This is a major drawback performance-wise when multiple rounds of computations are performed on the same data. To overcome this limitation (while retaining the scalability and fault tolerance of MapReduce) Spark introduced the concept of Resilient Distributed Dataset (RDD) which are in-memory data structures that can be kept in memory across a set of nodes that can be rebuilt if a partition is lost.

[1]Explained by one of the main minds behind Spark in this interview from May 2015 available here: `http://www.kdnuggets.com/2015/05/interview-matei-zaharia-creator-apache-spark.html`

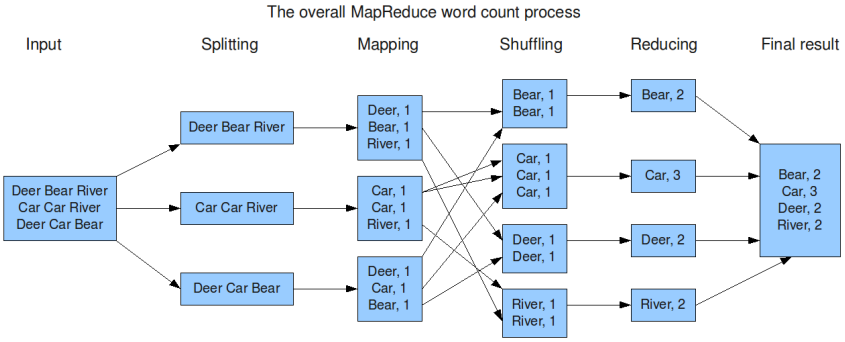


Figure 3.1: Different phases of performing a word count with the MapReduce framework.

Aside from the performance point of view this is another point where spark shines: the simplicity. As we said before, working with the MapReduce framework means adapting the computations to the MapReduce pattern. The programmer is required to think only in terms of basic operations of Map and Reduce. And the task of adapting every problem to this patter is not trivial. Common operations can require a significant amount of work. For example a simple word count program needs around 60 lines of code in Hadoop MapReduce while the same can be achieved in only one line in the interactive shell of Spark.

3.2 Spark Overview

By the Spark project we are referring to a set of different components each one specialized in different workloads. We can see the components in figure 3.2. At its core is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across a *computing cluster*. This core engine powers multiple higher-level components.

This design philosophy comes with several benefits. The obvious one is that the higher level components benefit from any improvement done in the lower layers. Another benefit is the cost of running the stack is minimized, instead of running a few independent software systems only one it is needed. This reduces significantly the cost of trying new types of analysis inside an organization.

As shown in the figure 3.2 we will only be using a few components of the whole Spark stack but we will briefly introduce each one of them:

Spark SQL

This component provides a SQL interface to Spark, allowing developers to perform SQL queries and programmatic data manipulations in the same application. It is the main way for Spark to work with structured data.

Spark Streaming

Spark Streaming provides an API for processing and manipulating live streams of data.

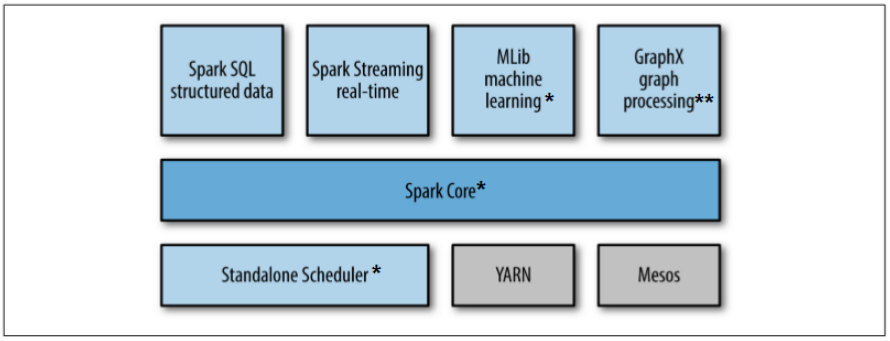


Figure 3.2: Components of Spark. Marked with * the components we actively used on this project. ** Is used for the implementation of the *EM* Optimizer of LDA. *YARN* and *Mesos* are not developed in the Spark project but appear on the picture because Spark can run on those two.

For example log files from a server or messages on social media. Under the hood, Spark Streaming retains the scalability and fault tolerance of the Spark Core.

MLib

MLib is Spark's machine learning library. It provides multiple types of machine learning algorithms, including classification, regression, clustering[2] among others. All of these algorithms are designed to scale out across a cluster. The Latent Dirichlet Allocation is implemented in this library as a part of the clustering algorithms.

GraphX

GraphX is a library for manipulating graphs and performing graph-parallel computations. Extends the Spark RDD API providing operations to manipulate graphs and a library of common graphs algorithms.

Spark Core

We will review in depth this component in the following sections, since this component contains the basic functionality of Spark, including components for task scheduling , memory management, fault recovery, interacting with storage systems, and more. Spark Core contains the main programming abstraction of Spark, the Resilient Distributed Datasets (RDDs).

Cluster Managers

Spark can run over a variety of cluster managers including Hadoop YARN and Apache Mesos. It also includes a simple cluster manager called the Standalone Scheduler. We will be using the Standalone Scheduler in the tests of chapter 4.

[2]Latent Dirichlet Allocation is considered a clustering algorithm in the MLib

3.3 Spark Programming Model

To use Spark the developer writes a *driver program* that controls the high-level flow of the application and launches operations in parallel (or calls to the API that launch operations). Each dataset is represented as an object and operations are invoked using methods on these objects. As part of the Spark Core component, the Spark API provides two main abstractions for the developer: the *Resilient Distributed Dataset* (RDD) and the *parallel operations*:

3.3.1 Resilient Distributed Dataset

A Resilient Distributed Dataset is a read-only collection of objects partitioned across the cluster that can be rebuilt if a partition is lost. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. The most usual ways to create an RDD are:

- Load from a file with the built in functions from a shared file system. For example loading a text file from HDFS (Hadoop Distributed File System).
- Parallelize a collection (e.g. an array from Scala) in the driver program.
- Transform another RDD. Although RDDs are immutable, we can apply transformations to an already existing RDD in order to create a new one that represents the new state of the previous RDD.

Most of the times when working with RDDs its elements do not exist in physical storage. An RDD only stores the information about how it was derived from others datasets so it can be computed at any time but it does not need to contain the computed data at all times. This allows the program to be able to reconstruct all the RDDs in case of failure.

The user can also control the persistence and the partitioning of an RDD. Persistence allows the developer to change if the RDD it is kept in memory so it can be reused faster in future operations. The partitioning controls how the dataset is spread across the cluster[3]. It is important to note that even if the persistence level is set to keep the RDD in memory it will spill to disk if there is not enough memory for it. It works this way so the system keeps working in case of failure of some nodes (or if a dataset is too big)[4].

As we commented before when talking about the motivation behind Spark, RDDs are best suited for batch applications where the same operation is done to all elements of a dataset since it can efficiently remember each transformation as one step. However this design makes RDDs less suitable for applications that make constant asynchronous updates like a storage system or a web crawler.

[3]We will comment more about how the partitioning affects real world performance and scalability in Chapter 4.

[4]The user can also request other persistence strategies, like storing an RDD only on disk or duplicate it across the cluster.

3.3.2 Parallel Operations

There are two types of operations in Spark: transformations, that construct a new RDD from a previous one, and actions, that compute a result based on an RDD. The main difference between transformations and actions comes from the way Spark computes RDDs. Although you can define new RDDs any time, transformations are computed in a lazy fashion, while actions launch a computation. Lazy fashion means that the transformations are not computed until the first time the RDD is used in an action. This may not make sense at first but it allows to make optimizations before executing the job.

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Figure 3.3: Most commons operations available on RDDs in Spark.[5]

3.4 Spark Architecture

In this section we will focus on the Spark Standalone Cluster that is the built-in cluster and comes with the default distribution. Most of the concepts are applicable for YARN and MESOS[6] but there may be some slight differences. Spark uses a master/slave architecture[7] where the driver program talks to the *clustermanager*, also referred as the *master*, that manages *workers* in which *executors* run. They can run all on the same computer or spread around a cluster.

In the figure 3.4 we introduce the main actors of the architecture:

Cluster Manager

The cluster manager is an external service for acquiring resources on the cluster. As we said before Spark has its own cluster manager but can also run in YARN and MESOS.

Driver

The driver is where the task scheduler is executed and the web UI Enviroment is hosted. The main responsabilites of the driver program is to coordinate the workers and the overall execution of tasks.

[5] Table from [9].
[6] MESOS and YARN are two of the most popular cluster manager.
[7] The slaves are called workers when talking about Spark.

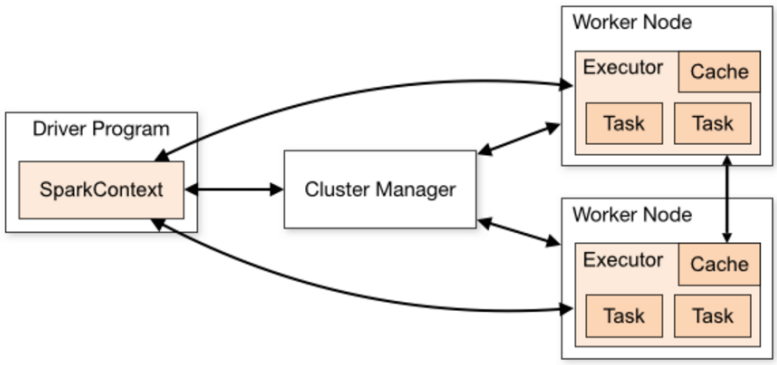


Figure 3.4: Spark architecture. [8]

Worker

Worker, also known as slaves, are the compute nodes in Apache Spark and host of the executors.

Executor

Executors are distributed agents that execute tasks. Executors are created when an Spark application is submitted to the driver and if nothing goes wrong they run for the entire lifetime of the application.

3.5 Spark Execution Model

In the previous section we have talked about the tools provided by Spark to the developer to build the applications. Now we will explain the internal process followed by Spark in order to execute said applications. This part is also important to understand the results we obtained in our scalability tests of Chapter 4 and to fine-tune the application to obtain the best performance possible.

At a high level, the process Sparks follows to execute a program has three steps:

- Create a DAG (Directed acyclic graph) of RDDs to represent the computation.
- Create a logical execution plan for the DAG.
- Schedule and execute individual tasks.

3.5.1 Create a DAG

This means detect the lineage of each RDD used in the application. Each RDD represents a node of the DAG and given two nodes A and B, A points to B if B is generated by performing an operation to A. Here comes into play the fact that RDDs are immutable so the resulting RDD Graph is always acyclic.

[8] Figure from the Spark documentation. Retrieved from: http://spark.apache.org/docs/latest/cluster-overview.html

This is easier to understand with a simple example, that we will also use for the other steps of the execution. We can think of a program that given a list of names counts the number of distinct names per "number of letters". The program would look like this [9]:

```
sc.textFile("hdfs:/Input.txt")
        .map( name => (name.charAt(0), name) )
        .groupByKey()
        .mapValues( names => names.toSet.size)
        .collect()
```

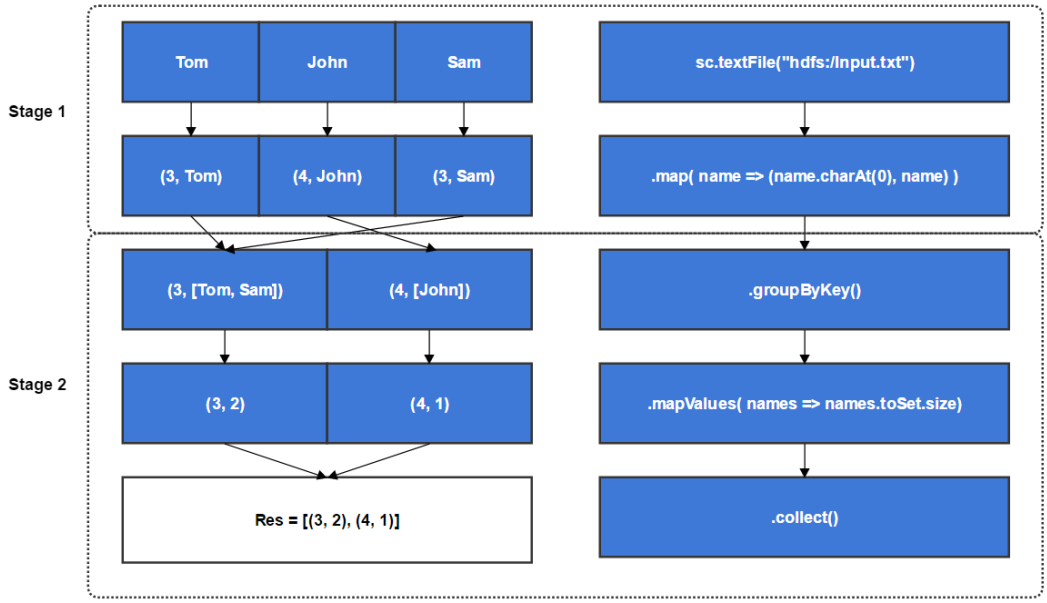


Figure 3.5: Flow of the example application with a sample dataset and DAG generated.

The DAG is a result of the written program and it shows the dependencies between RDDs. In this example the resulting DAG is simple because we are applying transformations in a linear fashion and we are not combining RDDs (see figure 3.5).

3.5.2 Create an execution plan

The DAG we defined on the previous point already gives us a way to execute the application, but we want to execute the application as efficiently as possible, for that we are going to "pipeline" as much as possible. By pipeline we mean to "fuse" operations together so we do not go multiple times over the data for no reason. We split the execution in stages based in the need to reorganize the data. Another way to see this is to pipeline while the result can be computed independently of any other data.

Following the example we fuse the two first operations into one, so at the same time we read each name, we apply the map function so that the overhead of multiple operations that are pipelinable is much lower. We can pipeline since to compute $(3, Tom)$ we do not need anything about John, but for the next step we do need the rest of the data so we can't pipeline any further. So Stage 1 will group the first two operations as shown in figure 3.4.

[9] The code is in Scala and assumes the SparkContext is already created as sc and a file in the hdfs with the dataset exists.

After this we do the same process again, we try to pipeline as much as possible and we see that we can group the rest of the application into Stage 2 [10].

3.5.3 Schedule tasks

Now that we have the logical execution plan we need to actually execute it. For that we split each stage into a set of tasks. A task is simply some data and a computation process that we can send to a node so it can be executed independently of everything else. The basic idea is that we need to execute all the tasks from one stage before we can move on to the next one.

In order to obtain one task we bundle an stage with a partition of data. Following the previous example we would bundle each block of the HDFS file and the Stage one of computation to obtain the different tasks.

Now we have identified the different tasks that we want to execute, we need to execute them across the cluster. Spark schedules the tasks in a pretty straight-forward manner. First it will try to execute the tasks on the node where the data is located. If the node is busy with another tasks it will wait for a set amount of time[11] and if the node is still busy it will execute on any free node and move the data to said node.

As we said before, the fact that separates one stage from another is the need to reorganize data or in other words the need to redistribute the data among partitions. This process is called $shuffle$. In the example this means that we need to group all the names that have the same number of letters. We are simulating that each partition has only one name so it goes only to one place but in reality each partition would contain multiple names and each name should go to the corresponding group according to its number of letters.

In traditional MapReduce frameworks this phase is often included as part of the Reduce phase. This is a point of constant development at the moment and it is probably one of the main points where Spark can improve its performance in the near future. Spark allows multiple ways to do this process and the one used it's determined by the value of a configuration parameter called $spark.shuffle.manager$. By default Spark uses a sort-based shuffle since version 1.2, before that it used a hash-based shuffle.

It is important to note before we start explaining how the shuffle works, that the data is written to disk between stages, we said before that in Hadoop's MapReduce the data is written to disk between each Map and Reduce stages. In Spark the data is only written between stages as part of the shuffle phase. Remember that in general writing data to disk is always slow and should be skipped when possible to improve performance.

Now we are going to explain the two main shuffle modes that Spark uses, the hash-based and the sort-based. There is a third one currently being developed available in experimental state called Tungsten Sort that we will not explain since it is not official yet. While we explain them we will follow the common MapReduce naming convention. We will call $mapper$ to the task that emits the data in the source executor and *reducer* to the task that consumes the target data into the target executor.

[10] The collect is a bit special since is not really an RDD. We can also pipeline it.
[11] More on this subject when we talk about the locality parameter in section 4.2

Hash-based Shuffle

This was the default shuffle mode prior to Spark 1.2[12]. The logic of this shuffler is pretty simple: each mapper task creates a separate file for each reducer and the mapper loops through the source partitions to calculate the target file (applying a hash function). The number of reducers is the number of partitions of the RDD. This shuffle is shown in the following figure.

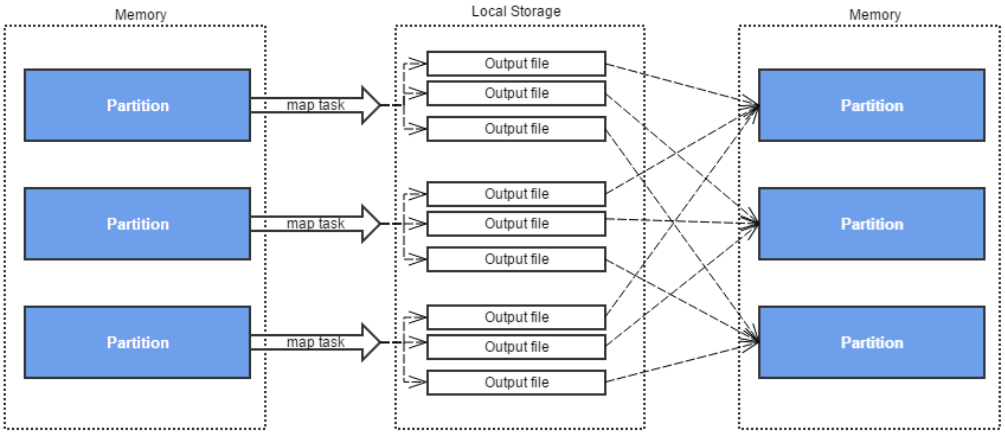


Figure 3.6: Hash-based Shuffle.

The main problem of this shuffle is the large amount of files it needs to write when there is a big number of partitions and executors. And the fact that it tends to use random I/O instead of sequential I/O which is in general up to 100x faster.

Sort-based Shuffle

Since version 1.2 Spark uses the sort-based shuffle. The actual implementation of this shuffle is a bit more complex in order to optimize the sort but the basic idea is that instead of creating one file for each reducer like we do in the hash-based shuffle, in the sort-based each mapper writes a single file ordered by reducer. The merging is only done when the data is requested by the reducer.

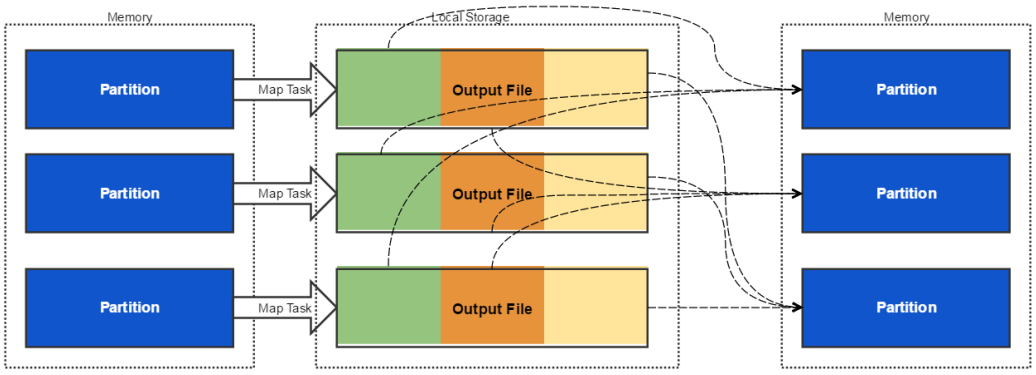


Figure 3.7: Sort-based Shuffle.

This shuffle has a clear drawback since sorting is harder than applying a hash so for smaller clusters this method may be slower. In order to avoid this spark has a configu-

[12]Spark 1.2.0 was released in December 18, 2014

ration parameter called *spark.shuffle.sort.bypassMergeThreshold* that sets the minimum amount of reducers to use the sort-based shuffle. This drawback is compensated by the smaller amount of files created and the fact that this method tends to use sequential I/O that as we already said is in general much faster than random I/O.

Chapter 4

Experimentation

In this chapter we will focus on testing the performance and scalability of the LDA implementation with the two possible inference algorithms we have seen in previous chapters. In the final part of the chapter we show the resulting topics obtained with LDA for both the New York Times and the BBC News Dataset.

4.1 Datasets

For this purpose we will be using a dataset created from a collection of articles of the New York Times by the Machine Learning Repository of the University of California Irvine [1]. It comes already as a bag of words, so the documents are already tokenized and we have the count value of each word in every article for the words that appear more than 10 times in the collection. The dataset contains around 300.000 articles with a vocabulary of 102.660 words. Making the total wordcount over 100.000.000. We will be using the number of CPU cores available for the scheduler as a variable to test the scalability.

The topic analysis of the New York Times dataset we can do is limited by the fact we only have the word count for each article but not the text of the article. In the last section of this chapter we introduce the BBC News Dataset[2] that is much smaller (2225 articles) but contains the whole article so we can actually compare the asigned topics with the content of the article.

4.2 Infrastructure

All the testing was realized in a Spark Standalone cluster of 4 workers each one with 6 GB of ram available and 4 CPU cores configured on top of OpenNebula[3]. OpenNebula is used to manage a shared-nothing non-dedicated cluster of four machines:

[1] The dataset can be found in [1]

[2] Available in [2].

[3] OpenNebula is an open source cloud computing platform for managing heterogeneous distributed data centers.

- Two of the machines have: two Intel Xeon E5620 2.40GHz processors (4 cores and 2 threads per core in each processor) and 24 GB of main memory.
- The third has: two Intel Xeon CPU E5645 2.40GHz processors (6 cores and 2 threads per core in each processor) and 24 GB of main memory.
- The fourth has: Xeon E5-2630 v2 2.60GHz processors (6 cores and 2 threads per core in each processor) and 32 GB of main memory.

The data was loaded from a single plain text file stored in HDFS of the same 4 computers. The used scheme is shown in the following figure (Figure 4.1):

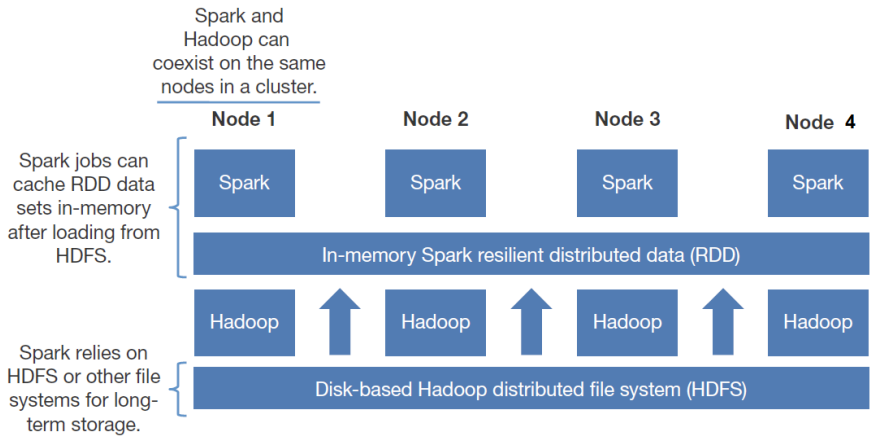


Figure 4.1: Scheme of the (virtual) architecture used during the testing.

The whole application used for the tests can be found as a part of the code of the project[4], but now we are going to explain how it works. Since we are only interested in testing the scalability of the LDA implementation all the application does is load a text file (that already comes with the word count for each document) stored in HDFS as a Spark RDD and transforms it to the format the LDA implementation takes as input and then executes 50 iterations of the LDA algorithm with the selected optimizer and number of cores.

4.3 Test Application

As we previously explained the Spark API is available in Scala, Python and Java. The application is written in Scala in order to obtain the best performance possible since that's the native language of Spark.

The application takes three command line parameters: the name of the file with the corpus, the LDA optimizer to use and the number of CPU cores. The number of CPU cores parameter it's only used for naming the spark application since we set this parameter when submitting the application to the server.

The first thing the application does is take those three parameters and set the name of the application in the spark context.

[4]Appendix D

```
val conf = new SparkConf().setAppName(args(0)+args(1)+args(2))
val sc = new SparkContext(conf)
```

Then we load the text file with the corpus as rdd and start doing the needed transformations in order to get the corpus in the format the LDA implementations needs.[5] It takes a RDD where each row is a document, and each document is one identification number and a vector of the word counts. The nth position of the vector represents the number of appearances of the nth word of the vocabulary in the document. The corpus we are using already comes as a bag of words with the three first lines as headers. Starting in the fourth row we have triplets $docID, wordID, wordCount$ separated by spaces.

First thing we do is load the plain textfile and remove the first three rows.

```
val corpusPath = "hdfs://reed:54310/user/ccortes/"+args(0)

// Load documents from text files.
//1 row of the RDD per line in the file.
val rawCorpus = sc.textFile(corpusPath)

//Remove the first 3 rows.
val firstRow = rawCorpus.first
val tempCorpus1: RDD[String] = rawCorpus.filter(x=> x!=firstRow)

val secondRow = tempCorpus1.first
val tempCorpus2: RDD[String] = tempCorpus1.filter(x=> x!=secondRow)

val thirdRow = tempCorpus2.first
val tempCorpus3: RDD[String] = tempCorpus2.filter(x=> x!=thirdRow)
```

Then we start the transformations: first we split each line in three $(docID, wordID, wordCount)$ and then keep the $docID$ as key and create an array with the $wordID$ and the $wordCount$. We also cast them to the appropriate type since we read them as strings and we are dealing with numbers.

```
val StringCorpus = tempCorpus3.map(x=> x.split(' '))

val doubleCorpus = StringCorpus.map(x=>(x(0).toLong,
        Array( x(1).toDouble, x(2).toDouble)))
```

The next step is group all the $(wordID, wordCount)$ that are from the same document (same $docID$), in the previous step we have the pairs $(wordID, wordCount)$ as an array so now it's easy to just concatenate the arrays.

```
val reducedCorpus = doubleCorpus.reduceByKey((x,y)=>x++y)
```

Right now each row of our RDD has the following format:

$$(Double, Array[Double]) = (docID, Array(WordID, WordCount, WordID, WordCount, ...))$$

[5] The `http://spark.apache.org/docs/latest/api/scala/#org.apache.spark.ml.clustering.LDA`

The LDA implementation needs each row of the RDD as follows.

$$(Long, Vector) = (DocID, Array(WordCount_1, ..., wordCount_V)$$

where $wordCount_i$ is the count of the word i of the vocabulary in the document *docID*. And we do this transformation with the following code:

```
val sparseCorpus = reducedCorpus.map{ case (docID, wordCount) =>
        val counts = new mutable.HashMap[Int, Double]()
        var i=0
        var j=0
        for (i <- 0 to (wordCount.size-1) by 2 ){

                counts(wordCount(i).toInt)= wordCount(i+1);
        }

        (docID, Vectors.sparse(102661, counts.toSeq))
        //102661 is the size of the Vocabulary
}
```

We go through each row (each *docID*) and create a HashMap that has *wordID* as key and the *wordCount* as value. Then we create the Sparse Vector the LDA implementation needs from the HashMap. The choice of Sparse Vector makes sense in our case since most of the documents will only use a small fraction of the vocabulary so we will be dealing with vectors with many zeros and Sparse Vector will save a lot of memory.

Now our corpus RDD has the correct format for the LDA implementation so all it is left is run the LDA algorithm:

```
val numTopics = 10
val lda = new LDA().setK(numTopics).setMaxIterations(50)
lda.setOptimizer(args(1))
sparseCorpus.persist(StorageLevel.MEMORY_AND_DISK)
val ldaModel = lda.run(sparseCorpus)
```

We run LDA for 50 iterations [6] and with the optimizer we got from the command line arguments. The number of topics have no impact in the performance, we choose 10 topics. It's also worth noting the *StorageLevel.MEMORY_AND_DISK*, that means that it will save the RDD to the disk if it runs out of memory.

4.4 Results

In this section we will show and analyze the results we obtained on the tests [7], the ideal result would be to reduce the execution time by half every time we double the number of cores. That would mean that we are achieving perfect scaling with the computational power (represented in our case by the number of cpu cores used). This would be ignoring the fact that we can face memory bottlenecks, when using multiple workers we are

[6] The documentation recommends between 50 and 100 iterations.

[7] All the results are available in Appendix A.

multiplying the memory as well as the number of CPU cores. We can anticipate that the memory will not be an issue when executing with the online optimizer since unlike the em optimizer we do not need to have the entire corpus on memory.

All the results were executed without specifying the number of partitions, we observed in the executions that the RDD had 8 partitions in all cases which makes sense since the dataset weights 957 MB and the hadoop block size is of 128 MB. ($957/128 \approx 7.47$ so we need 8 hadoop blocks and each block is taken as a partition by default).

We executed three times with each setting and since all the results were quite similar we assumed that the three executions were enough for the analysis we aim to do. All the times displayed are in seconds and we will use the median to summarize the results instead of the average due to the low number of attempts.

The first thing we need to explain before we start analyzing the results is that the cluster computers are not used only for the Spark application so some variation in the results is expected depending on how heavily the cluster is being used at the time of the testing. To realize the tests we execute the application we have explained with all the different parameters we are interested to analyze multiple times. We will call attempt to each iteration of executions using all the combinations of parameters. To execute all the parameters we created a simple Python script[8] that loops through all the parameters and submits the apps to the cluster. Once all the applications are finished we get the execution times by calling the Rest Api[9] that Spark provides with another Python script[10].

4.4.1 Scalability of LDA with EM Optimizer

When executing with the EM optimizer we obtained the following execution times:

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160310_nolocality	4063.39	3014.36	2127.92	4230.84	1671.04
20160307_nolocality	4313.74	3054.02	1799.09	4610.75	2114.60
20160306_nolocality	4370.40	2956.01	1879.17	4210.07	1922.43
median	4313.74	3014.36	1879.17	4230.84	1922.43

We can see that when going from 1 CPU core to 4 CPU cores the results are more or less what could be expected. From 1 to 2 cores the execution time (on the median) reduces a 30.12% and from two to four it reduces to 37.66%. But then when between 4 and 8 cores something is not working as expected, the execution time with 8 is even longer than with 1 core.

After some investigation we detected the reason behind this abnormal behavior. In the monitoring web UI we could see that with the others numbers of CPU cores all the tasks had the value *PROCESS_LOCAL* in the field *Locality Level*, but when executing with 8

[8] Available with the code of this project. The script is called Call.py

[9] The Rest Api documentation can be found here: http://spark.apache.org/docs/latest/monitoring.html#rest-api

[10] Also available with the code of this project. The script is called Load.py

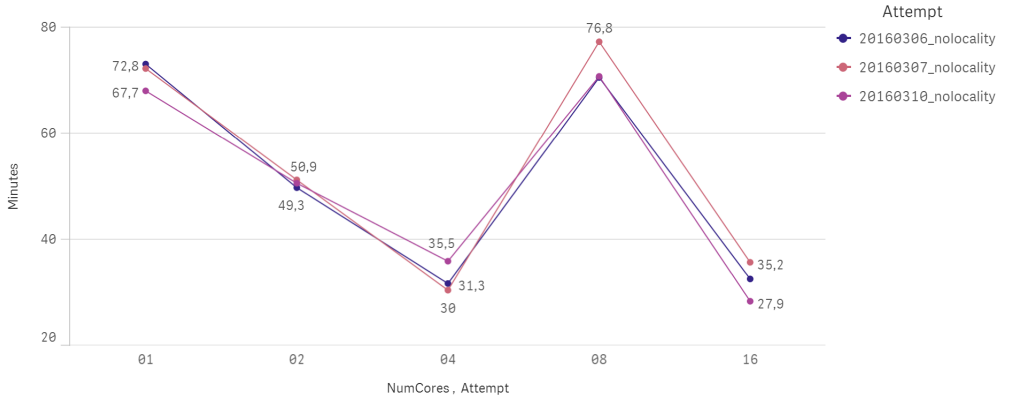


Figure 4.2: Line chart for the results with EM optimizer.

CPU cores some tasks were taking much longer to finish that the rest and those tasks had the value *ANY* in the *Locality Level* field. We also observed this situation for executions with 16 CPU cores but the number of tasks where this happened was much lower so the affect in the performance is much lower.

The Locality Parameter

That abnormal behavior we commented was due to the location of the data. As we already said the data is stored in HDFS so the data is distributed along the cluster. In all cases we need to have the data in the executor node before doing any calculations with it and this was the problem we were facing. The Spark Scheduler was assigning tasks to executors that did not have the data related to the task, so before doing any calculations the data had to be sent to the executor and that was taking much longer that the calculation to be done.

Following what we explained about the Spark scheduling in Chapter 3, what Spark typically does is wait a bit hoping that a busy CPU core with the data frees up when assigning a task. Once that period expires, it starts moving the data to the free CPU. The waiting period can be configured with a parameter called *spark.locality.wait*. We configured this parameter with a high value in order to always wait until the CPU with the data frees up. The results obtained with this new value for the parameter are in the following table:

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160404	4692.07	2634.98	2051.21	1792.90	1455.66
20160321	4581.03	3162.55	1944.10	1765.16	1567.20
20160316	5448.16	3677.77	2420.31	1717.25	1304.69
median	4692.07	3162.55	2051.21	1756.16	1455.66

The obtained results when setting up the locality parameter trying to solve the issue we have talked about are summarized in figure 4.3. We can observe that by setting up the locality parameter we solved the problem we had in the 8 and 16 CPU cores executions but the improvement wasn't as good as we could guess by the improvement we obtained

when going from 1 to 4 cores.

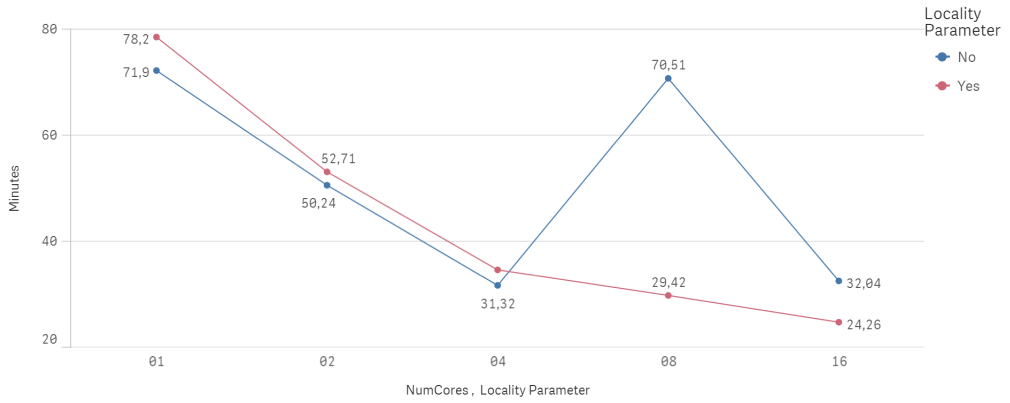


Figure 4.3: Line chart for the median of the execution time of the attempts with the em optimizer and the locality parameter configured to wait vs the attempts without it.

4.4.2 Scalability of LDA with Online Optimizer

When executing with the Online optimizer we obtained the following execution times:

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160310_nolocality	2886.03	2272.83	1156.68	970.81	821.10
20160307_nolocality	2706.07	2079.01	1277.14	859.81	850.23
20160306_nolocality	2943.42	2135.88	1340.70	954.89	824.46
median	2886.03	2135.88	1277.14	954.89	824.46

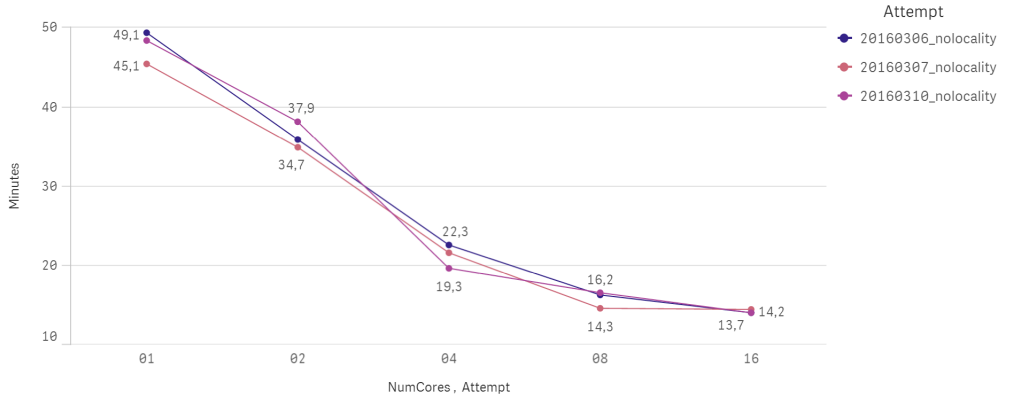


Figure 4.4: Line chart for the execution time for each attempt with the online optimizer.

In this case the application performs as could be expected. Obtaining an improvement (on the median) of 25.99% when going from one to two cpu cores, 40.20% from two to four, 25.23% from four to eight and 13.66% from eight to 16 cpu cores.

Locality Parameter

Remember that when executing the Variational Inference Algorithm we need to go through the entire dataset, when executing the Online Variational Inference we only need small portions of the dataset to do the update, so the location of the data is not as important as in the EM algorithm. Future work should examine why is that behavior in the online optimizer

We executed with the locality parameter like we did for the EM optimizer but in this case we see that it had no effect in the performance.

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160404	2996.05	1914.50	1811.40	1232.54	1075.42
20160321	3030.61	2097.13	1361.01	1039.76	1054.15
20160316	3224.53	1662.66	1438.19	1172.14	1005.06
median	3030.61	1914.50	1438.19	1172.14	1054.15

4.4.3 Resulting topics

Once we have run the algorithm for the set number of iterations we have created a LDA Model, to extract the topics we invoke the method $describeTopics(n)$ to the LDA Model, where n is the number of words per topic we want.

The obtained topics after one execution are shown in the following tables[11]. It is important to note that the LDA algorithm does not return the same results for each execution since it incorporates some randomness to the initialization process so the results may be a bit different each time we execute. This results are when executing with $k = 10$, so we obtain 10 topics. For each topic we show the five words more relevant.

- Results with online optimizer:

Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
car	king	official	company	food
driver	con	zzz_bush	million	cup
tour	una	government	companies	minutes
race	zzz_india	president	percent	add
hotel	mas	campaign	program	restaurant

Topic 6	Topic 7	Topic 8	Topic 9	Topic 10
book	water	percent	team	show
horse	building	market	game	book
race	found	stock	season	film
horses	plant	million	player	family
zzz_kentucky_derby	air	company	play	com

- Results with em optimizer:

[11] In Appendix B we show the weights obtained for each word

Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
team	official	official	com	water
game	zzz_united_states	attack	web	food
season	system	death	car	room
player	problem	police	site	minutes
play	zzz_u_s	drug	computer	small

Topic 6	Topic 7	Topic 8	Topic 9	Topic 10
percent	zzz_bush	country	show	school
company	president	history	film	student
million	campaign	zzz_american	women	home
companies	political	country	movie	family
market	zzz_george_bush	zzz_new_york	book	told

We can see that both algorithm obtain pretty similar topics and the differences can be attributed to the randomness of the algorithm. Future work could study these difference through a topic coherence metric[12] This makes sense since both algorithms maximize the same objective function as we saw in chapter 3. The shown topics are the results of a first execution of the algorithm, in order to improve the results we could run again the algorithm deleting from the dataset all the stopwords. For example from the results we obtained we could remove like *con* or *una* (Spanish stopwords) and execute again to obtain better topics.

4.5 Analyzing the BBC News Dataset

As we commented earlier, in the New York Times dataset we have the words counts but not the actual content of the article so we could show the obtained topics but we can not show the assigned topics to an article. In the BBC News Dataset we have 2225 articles corresponding to stories in five topical areas from 2004-2005: business, entertainment, politics, sport and tech. In the previous section our main focus was to analyze the performance of the algorithm, but now we are interested in analyzing the obtained topics and show the possibilities the LDA Model gives us to gain insights from the document collection.

The whole application can be found in Appendix D but it is quite similar to the one we showed for the New York Times Dataset until we run the LDA except that now we need to obtain the word count for each document before executing the algorithm. After running the LDA, we get the topics assigned to each document with the method *topicDistributions* of the LDAModel.

In order to obtain the best topic assignment we removed the 20 most common words in the dataset before executing the algorithm. Then we executed the algorithm for a first time and observed some words that appeared in the topics that were not useful for the analysis we are doing. The words we decided to remove were: *last*, *first*, *there*, *other*, *such*, *many*, *some*, *what*, *while*, *into*, *when*, *back*, *those*, *still*.

Then we executed the algorithm again (with $k = 5$ to obtain five topics) and obtained the following topics:

[12]The topic coherence metric is introduced here: http://dirichlet.net/pdf/mimno11optimizing.pdf

Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
film	year	against	government	mobile
best	sales	world	told	technology
show	market	england	should	make
number	company	best	labour	used
music	economic	over	public	digital

In Appendix C we can find a D3 Bubble Chart visualization of this table that also shows the weight of each word in the topic. The documents of this dataset come already classified in 5 categories: business, entertainment, politics, economics and sports and we can see a clear relation between the obtained topics and these categories, but it's important to remember the fact that LDA assumes that each text comes from multiples topics and returns the weight of each topic for each document. We can see the relation between the BBC categories and the assigned topics in the following table that shows the average weight of the topics for the documents of each of the BBC categories:

Avg. Assigned Topic / BBC Category	Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
business	0.10	0.51	0.10	0.15	0.13
entertainment	0.48	0.12	0.15	0.13	0.11
politics	0.11	0.13	0.10	0.55	0.11
sport	0.12	0.10	0.54	0.13	0.11
tech	0.18	0.11	0.08	0.10	0.52

From now on we will assume that Topic 1 refers to entertainment, Topic 2 refers to business, Topic 3 refers to sport, Topic 4 refers to politics and Topic 5 refers to tech.

Since we have the weight of each topic for each document we can assign to each document the topic with the biggest weight and use LDA as a classifier. The following table shows the number of documents we assign to each topic

Num. Doc. Assigned / BBC Category	Total	Topic 1 (entert.)	Topic 2 (business)	Topic 3 (sport)	Topic 4 (politics)	Topic 5 (tech)
business	510	5	467	1	25	12
entertainment	386	356	2	8	12	8
politics	417	7	16	1	391	2
sport	511	0	2	506	3	0
tech	401	69	6	2	6	318

So 91.60% of the time the assigned topic by LDA is the same as the BBC category. The documents where the assigned topic does not match the BBC category show some interesting mixture of topics that highlight the difficulty of Topic Modeling. For example we can review the article *entertainment*\001.*txt*:

Gallery unveils interactive tree

> A Christmas tree that can receive text messages has been unveiled at London's Tate Britain art gallery.
>
> The spruce has an antenna which can receive Bluetooth texts sent by visitors to the Tate. The messages will be "unwrapped" by sculptor Richard Wentworth, who is responsible for decorating the tree with broken plates and light bulbs. It is the 17th year that the gallery has invited an artist to dress their Christmas tree. Artists who have decorated the Tate tree in previous years include Tracey Emin in 2002.
>
> The plain green Norway spruce is displayed in the gallery's foyer. Its light bulb adornments are dimmed, ordinary domestic ones joined together with string. The plates decorating the branches will be auctioned off for the children's charity ArtWorks. Wentworth worked as an assistant to sculptor Henry Moore in the late 1960s. His reputation as a sculptor grew in the 1980s, while he has been one of the most influential teachers during the last two decades. Wentworth is also known for his photography of mundane, everyday subjects such as a cigarette packet jammed under the wonky leg of a table.

This article is from the entertainment BBC category and talks about a bluetooth Christmas tree that can receive messages so it is not a unreasonable to classify it as a tech article like LDA did. The LDA weights are shown in the following table:

Avg. Assigned Topic / Document	Topic 1 (entert.)	Topic 2 (business)	Topic 3 (sport)	Topic 4 (politics)	Topic 5 (tech)
entertainment\001.txt	0.25	0.13	0.12	0.12	0.38

Chapter 5

Conclusion

This chapter presents the conclusions of this work. It is organized in two sections: Conclusions and future work. The former summarizes the project's goals and the latter exposes some ways this project could be extended.

Conclusions

In this work we presented the concept of Probabilistic Topic Modeling and explained the Latent Dirichlet Allocation from a theoretical point of view. Once we had an strong knowledge of how the algorithm worked we introduced Apache Spark and all the concepts we needed in order to test scalability of the Latent Dirichlet Allocation implementation and understand the results we obtained.

On the subject of the scalability we saw how adding CPU power may not be worth the cost in some cases since the improvement obtained was small at best when going from 8 to 16 cores. The tests highlighted the importance of understanding the details of how Spark works since we obtained a huge performance boost just by changing a configuration parameter.

Regarding the Topic Modeling results we could observe that LDA gave us some good results with the BBC News dataset where in almost 92% of the documents we assigned the same topic as the BBC and reviewing the errors we could see that there were some reasonable missclassifications (with the source code are all the weights of each topic for all the documents).

Future Work

The LDA already gave us some good results but it can be improved to reduce the impact of the judgment calls the developer needs to do, by that we mean mainly choosing the number of topics and choosing the stopwords. Algorithms that extend the LDA model to scenarios where the number of topics is not known a prior have been developed like the HDP (Hierarchical Dirichlet Process) where another Dirichlet distribution is added to capture the uncertainty in the number of topics. The stopwords problem is harder to solve and appears when dealing with natural language, some lists of standard stopwords exists but specific problems or datasets may need different stopwords.

Related to the scalability tests some more fine-tuning could be done around the num-

ber of partitions and see how this affects the performance. This would be more relevant in scenarios with even bigger datasets than the one we used and bigger clusters of computers.

Lastly, during this work when talking about the quality of the obtained topics we always skipped defining what a good topic assignment is and how to measure it. We used the BBC News dataset that had the articles already classified in five categories to compare with the most important topic in each document according to the LDA result. The concept of perplexity is often used in this context to evaluate how well an algorithm works but also other concepts like the topic coherence metric have been introduced. Some future work could include adding the perplexity or the topic coherence metric to the tests realized in this work, this would also allow us compare the Online Variational and the Variational algorithm.

Bibliography

Datasets

[1] M. Lichman, . *UCI Machine Learning Repository*, University of California, School of Information and Computer Science (2013). Retrieved from http://archive.ics.uci.edu/ml.

[2] D. Greene and P. Cunningham, *Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering*, Proc. ICML 2006. Retrieved from http://mlg.ucd.ie/datasets/bbc.html

Probabilistic Topic Models

[3] D. Jurafsky, *Language Modeling: Introduction to N-grams*, Lecture Notes from Stanford University. Retrieved from https://web.stanford.edu/class/cs124/lec/languagemodeling.pdf

[4] K. Nigam, A. McCallum, T. M. Mitchell, *Semi-Supervised Text Classification Using EM*, Book chapter in O. Chapelle, A. Zien and B.Scholkopf. Semi-Supervised Learning. MIT Press: Boston. 2006. Retrieved from https://people.cs.umass.edu/ mccallum/papers/semisup-em.pdf

[5] T. Hofman, *Probabilistic latent semantic indexing*, Proceedings of the Twenty-Second Annual International SIGIR Conference, 1999. Retrieved from https://arxiv.org/ftp/arxiv/papers/1301/1301.6705.pdf

[6] D. M. Blei, A. Y. Ng, M.I. Jordan, *Latent Dirichlet Allocation*, Journal of Machine Learning Research 3 (2003) 993-1022. Retrieved from https://www.cs.princeton.edu/ blei/papers/BleiNgJordan2003.pdf

[7] D. M. Blei *Variational Inference*, Lecture Notes from Princeton University. Retrieved from https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf

[8] M. D. Hoffman, D. M. Blei, F. Bach *Online Learning for Latent Dirichlet Allocation*, Advances in Neural Information Processing Systems 23 (NIPS 2010). Retrieved from: https://www.cs.princeton.edu/ blei/papers/HoffmanBleiBach2010b.pdf

Apache Spark

[9] H. Karau, A. Konwinski, P. Wendell, M. Zaharia *Learning Spark, Lighting Fast Data Analysis*, O'Reilly 2015.

[10] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, *Spark: Cluster Computing with Working Sets*, HotCloud 2010, June 2010.

[11] A. Davidson, *A Deeper Understanding of Spark Internals* , Youtube, 17 July 2012. Retrieved from https://www.youtube.com/watch?v=dmL0N3qfSc8.

[12] S. Farooqui, *Advanced Apache Spark* , Youtube, 10 April 2015. Retrieved from https://www.youtube.com/watch?v=7ooZ4S7Ay6Y.

[13] M. Zaharia, *Introduction to AmpLab Spark Internals* , Youtube, 21 Dec 2012. Retrieved from https://www.youtube.com/watch?v=49Hr5xZyTEA

[14] *Cluster Mode Overview - Spark 1.6.0 Documentation*. Apache Spark Documentation. Retrieved from https://spark.apache.org/docs/1.6.0/cluster-overview.html

[15] A. Grishchenko, *Spark Architecture Shuffle: Distributed Systems Architecture*. Retrieved from https://0x0fff.com/spark-architecture-shuffle/

[16] J. Laskowski, *Mastering Apache Spark*. Retrieved from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-architecture.html

Appendices

Appendix A: Performance Results

Here we show all the results of the tests. For each attempt we show the duration of the execution in seconds. We use the date of the execution and the locality parameter as name of the attempt.

- Executions times with EM optimizer:

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160310_nolocality	4063.39	3014.36	2127.92	4230.84	1671.04
20160307_nolocality	4313.74	3054.02	1799.09	4610.75	2114.60
20160306_nolocality	4370.40	2956.01	1879.17	4210.07	1922.43
20160404	4692.07	2634.98	2051.21	1792.90	1455.66
20160321	4581.03	3162.55	1944.10	1765.16	1567.20
20160316	5448.16	3677.77	2420.31	1717.25	1304.69

- Executions times with online optimizer:

Num. of Cores / Attempt	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
20160310_nolocality	2886.03	2272.83	1156.68	970.81	821.10
20160307_nolocality	2706.07	2079.01	1277.14	859.81	850.23
20160306_nolocality	2943.42	2135.88	1340.70	954.89	824.46
20160404	2996.05	1914.50	1811.40	1232.54	1075.42
20160321	3030.61	2097.13	1361.01	1039.76	1054.15
20160316	3224.53	1662.66	1438.19	1172.14	1005.06

Appendix B: Topics of the New York Times Dataset

Topics obtained with the online optimizer

Topic 1	
word	weight
car	0.009 642
driver	0.004 546
tour	0.003 830
race	0.003 470
hotel	0.002 552

Topic 2	
word	weight
king	0.014 357
con	0.011 831
una	0.008 733
zzz_india	0.007 676
mas	0.006 535

Topic 3	
word	weight
official	0.005 037
zzz_bush	0.004 869
government	0.004 491
president	0.003 963
campaign	0.003 544

Topic 4	
word	weight
company	0.007 943
million	0.005 593
companies	0.004 975
percent	0.004 455
program	0.003 486

Topic 5	
word	weight
food	0.007 930
cup	0.007 354
minutes	0.005 077
add	0.004 736
restaurant	0.003 898

Topic 6	
word	weight
book	0.007 856
horse	0.007 138
race	0.006 850
horses	0.005 829
zzz_kentucky_derby	0.004 233

Topic 7	
word	weight
water	0.003 381
building	0.003 106
found	0.002 391
plant	0.002 365
air	0.002 318

Topic 8	
word	weight
percent	0.016 931
market	0.009 638
stock	0.008 323
million	0.007 395
company	0.007 145

Topic 9	
word	weight
team	0.012 621
game	0.011 434
season	0.010 134
player	0.008 520
play	0.007 407

Topic 10	
word	weight
show	0.003 870
book	0.003 383
film	0.003 148
family	0.002 551
com	0.002 344

Topics obtained with the em optimizer

Topic 1	
word	weight
team	0.017 653
game	0.014 307
season	0.013 258
player	0.010 972
play	0.009 850

Topic 2	
word	weight
official	0.006 243
zzz_united_states	0.005 665
system	0.004 717
problem	0.004 621
zzz_u_s	0.004 557

Topic 3	
word	weight
official	0.007 665
attack	0.007 271
death	0.005 394
police	0.005 244
drug	0.005 067

Topic 4	
word	weight
com	0.014 326
web	0.008 730
car	0.008 128
site	0.007 804
computer	0.007 511

Topic 5	
word	weight
water	0.006 508
food	0.005 847
room	0.004 476
minutes	0.003 966
small	0.003 722

Topic 6	
word	weight
percent	0.018 414
company	0.015 521
million	0.014 090
companies	0.010 109
market	0.009 593

Topic 7	
word	weight
zzz_bush	0.011 063
president	0.010 731
campaign	0.008 839
political	0.007 370
zzz_george_bush	0.006 873

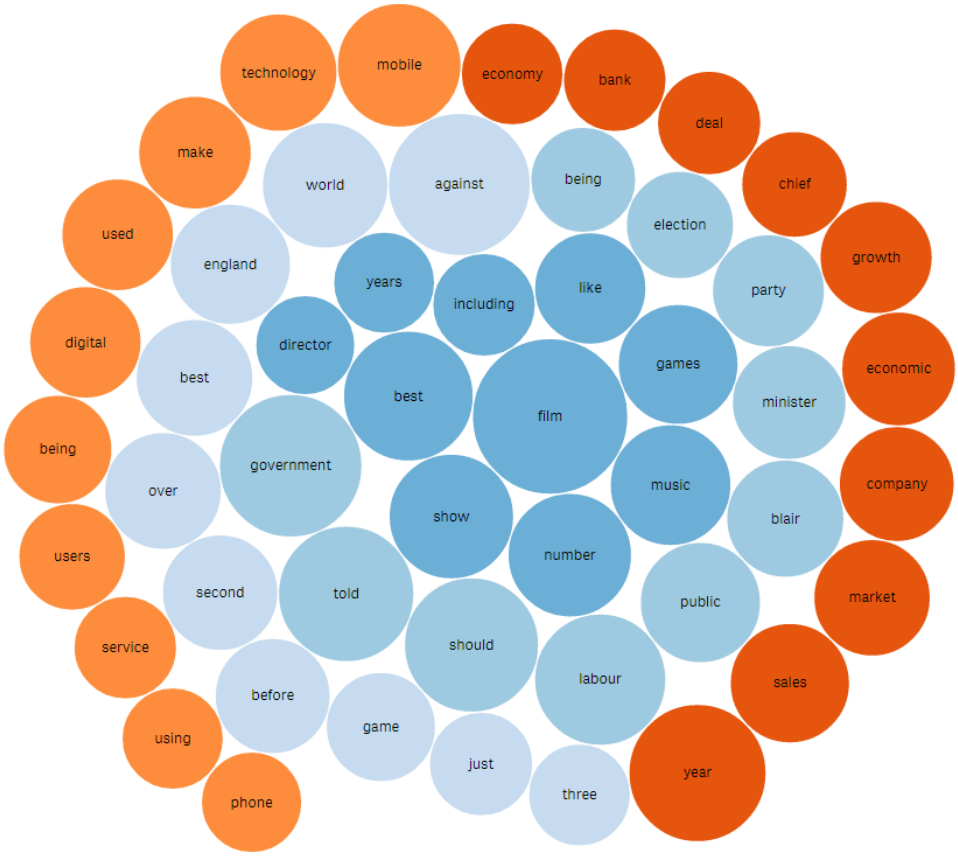
Topic 8	
word	weight
country	0.005 700
history	0.005 478
zzz_american	0.004 651
century	0.004 272
zzz_new_york	0.004 244

Topic 9	
word	weight
show	0.010 434
film	0.007 022
women	0.005 015
movie	0.004 917
book	0.004 606

Topic 10	
word	weight
school	0.013 288
student	0.007 637
home	0.006 321
family	0.006 253
told	0.005 974

Appendix C: Topics of the BBC News Dataset

We use a D3 Bubble Chart visualization to show the obtained topics with LDA on the BBC news dataset. Each color represents a topic, and the size of the bubble represents the weight of the term in the topic.



Appendix D: Source Code

Along with this document a source code package is provided where we can find the code used during the work. It is structured in three folders:

- **TestApp**: here we can find the scala application used for the scalability analysis. It needs to be compiled with the command *sbt package* and then submitted to the cluster with three parameters: the path to the bag of words file, the name of the optimizer to use and the number of cores. This application is explained in detail in chapter 4.
- **Script**: in this folder we can find two python scripts,the one used for submitting the apps to the cluster sequentially the script used to export the obtained results to a txt file by connecting to the Spark Rest API. The scripts are executable in the Unix shell if python is installed in: */usr/bin/env*.
- **BBC**: here we provide the scala code used to realize the topic analysis of the BBC dataset. We used the *spark−shell* to execute this code so there is no need to compile anything, only execute the code in the *spark−shell*. This code writes the resuls to plain txt files. The three folder with the plain txt exported are included. The *documents* folder contains the files with the topics assigned to each document. The *ids* folder contains the files with the id for each document. And the *topics* folder contains the files with words for each of the topics. The reason behind exporting a different file for each document was to avoid synchronization issues since we are exporting from an RDD.